Correctness by Construction

Peter Amey, Praxis High Integrity Systems [vita[1]]

2006-12-05 L3 / M[2]

Correctness by Construction (CbyC) is a radical, effective, and economical method of building software with demonstrable integrity for security- and safety-critical applications. CbyC combines the best parts of two superficially unlikely bedfellows: formal methods and agile development. For example, we take from the former *precise notations* and from the latter *incremental development*.

Praxis[3] has evolved CbyC over the last 12 years and used the approach to produce software in an industrial environment with extremely low defect rates; rates are fewer than 0.05 defects per 1,000 lines of code, with good productivity, up to around 30 lines of code per person day averaged over the development lifecycle. For examples of the use of CbyC in a security environment, see [Hall 02[4]] and [Barnes 06[5]].

CbyC is based on three simple principles:

1. Make it very hard to introduce errors.
2. Ensure that errors are removed as close as possible to the point of introduction, as they will be made despite item 1.
3. Generate evidence of fitness for purpose throughout development as a natural by-product of the process (because *showing* that the developed system is secure or safe is often harder than *making* it so).

These principles may sound like a rather obvious “counsel of perfection,” but they can be readily achieved with the right notations, tools, processes, and mindset.

Perhaps the neatest and quickest way of grasping the novelty of CbyC is to consider its linguistic near opposite: *construction by correction* (i.e., *build and debug*), which is still the way the majority of software is developed today.

In contrast to build and debug, CbyC seeks to produce a product that is initially correct. Testing becomes a demonstration of correct functionality rather than the point where debugging can begin. CbyC is a natural fit to the goals of Build Security In; both emphasize the need to ensure that a system is developed integrating required properties rather than just retrospectively examined for those required properties.

CbyC is technical approach to software development, which is highly compatible with the process principles of PSP/TSP[6]. Tentative, small-scale evidence shows that CbyC combined with PSP/TSP can result in even lower defect rates. The technical approach of CbyC can complement PSP to provide high process yields.

The rest of this article will expand on the principles of CbyC and how they can be implemented in practice.

Building Blocks

How do we achieve the goals of error prevention backed up by early error detection (other than by employing superhuman engineers who never make mistakes)? We do so by employing the following techniques, which are the building blocks of the CbyC process.

1. http://buildsecurityin.us-cert.gov/bsi/about_us/authors/620-BSI.html (Amey, Peter)
3. http://www.praxis-his.com/
4. #dsy613-BSI_hall02
5. #dsy613-BSI_barnes
6. http://www.sei.cmu.edu/tsp/

Sound Notations

A key way of avoiding the introduction of errors is the use of sound, formal notations for all deliverables. For example, specification notations such as Z[7] make it impossible (or at least very hard) to write ambiguous specifications. Ambiguity is a key source of errors because is allows members of the development and validation teams to interpret requirements differently. Formality in specifications is also a key tool in uncovering imprecise or contradictory requirements. Trying to express a user's request that "I want it to do X" in a precise mathematical notation is highly effective in determining that X's meaning is less clear than hoped. The use of formal notations thus contributes to both error prevention (because of the elimination of ambiguity) and to early error detection (because requirement inconsistencies are exposed during the specification process rather than during coding or testing or when in service).

CbyC, in its purest form, takes the use of rigorous, unambiguous notations a step further into programming language. Most commonly used programming languages allow the construction of programs of uncertain meaning. Unpredictable behavior can arise from factors such as freedom of sub-expression evaluation order in the presence of side effects or variable aliasing. These problems affect the ability to reason about source code and lead to overdependence on *observing* the behavior of code during testing rather than *predicting* the behavior of code during construction. Our language of choice is SPARK[8], a formally specified, annotated subset of Ada that was designed expressly for fast and deep static analysis. The use of SPARK *prevents* the introduction of a number of possible errors (e.g., buffer overflows, use of uninitialized variables). This form and use of static analysis is quite different from the use of *code scanning* tools, which use heuristic approaches to find potentially troublesome constructs. These weaker forms of static analysis, although better than nothing, have significant disadvantages; typically, they have high false-alarm rates, fail to detect the most difficult problems, and often can be deployed only near the end of the development process when change is hardest and most expensive.

CbyC *requires* precise notations. If one adopts the ideas outlined here without the use of precise specifications and verifiable programming languages, all that remains are good ideas of agile movement (e.g., test-driven development). The *combination* of these ideas and the best formal methods makes CbyC special.

We can capture the idea of using precise notations with the sound bite ***write right***.

Strong Validation

Because CbyC uses precise, unambiguous notations, it becomes possible to use strong, tool-supported methods to validate the deliverables of each development stage. For example, one can prove that the formal specification has certain required security properties, that the source code is free from run-time errors, and that the source code correctly implements key properties of the specification. Ensuring that each refinement step from user requirements to object code faithfully translates its input contributes to the principle of early error detection.

We can capture this idea with the sound bite ***check here before going there***.

Incremental Development

Here CbyC focuses on two complementary ideas: minimizing the semantic gaps between artifacts and minimizing the time when the system cannot be built and run because of incomplete development work.

The first of these ideas is essential to early error detection. If the semantic gap between artifacts is large (e.g., object code generated directly from a weakly-defined graphical design notation), the ability to predict behavior is lost. Strong validation cannot apply, and only testing and debugging are left.

The second idea, now commonplace in agile development, suggests that features be incrementally added so that each fully functions before adding another; although it would be incomplete, the system could be

7. http://vl.zuser.org/
8. http://www.sparkada.com/

built upon and run at all times during development. In CbyC, the first build is a complete skeleton of the system with all its interfaces and communication mechanisms is place; real functionality is incrementally achieved. The system can be tested and demonstrated early on, which is an important confidence-building measure.

The combination of small semantic gaps, precise notations, and strong validation achieves the third principle of generating certification evidence as a by-product of the development process.

We can capture this idea with the sound bite ***step, don't leap***.

Avoidance of Repetition

If ambiguity is the major cause of bugs, then repetition is a close second. If something is specified, documented, or implemented in more than one place, it is nearly certain that the various descriptions will become inconsistent. A special variant of Murphy's law should cover this phenomenon! Many development methods, especially those that attempt a systematic, waterfall approach, can fall easily into this trap: top-level designs are complemented by detailed and code module designs, each of which describes the same thing at increasing levels of detail. In CbyC, more detailed design documents are written only when needed (e.g., the semantic gap would be too large without an extra step). They contain only *additional* information needed and do not repeat information found in earlier deliverables.

A particularly important example of the avoidance of repetition is CbyC's separation of the ***software specification*** from the ***high-level design***. The former is concerned only with describing what the software will do, in terms of inputs, outputs, and state changes. The latter is concerned only with the how the system will be structured and architected to meet requirements such performance, safety, and security. The design does not repeat any information from the specification and, in practice, the two can be developed in parallel.

We can capture this idea with the sound bite ***say something once, why say it again?***

Striving for Simplicity

Because CbyC is based on the ideas of strong validation and ability to show in advance that a system *will be* fit for purpose, it follows that easily validated software must be designed and produced. The code should be simple and directly traceable to the specification. Because the test cases are also obtained from the specification, the process of verification is simplified.

Striving for simplicity is hard work but is worth the effort. Increasingly intricate software development methods seem to relish complexity for its own sake; unthinking use of object orientation is a prime suspect in this area! Increased production speed gained by using prebuilt library data structures, object factories, and wizards will be nullified when attempting to demonstrate fitness for purpose.

We simply say ***KISS***.

Managing Risk

When faced with a complex task, the natural tendency is to start with the parts you understand with the hope that the less obvious parts will become clearer with time. CbyC consciously reverses this. As risk and potential bugs hide in the most complex and least understood areas, these areas should be tackled first. Another reason for tackling uncertainty early is that freedom for maneuver tends to decrease as the project progresses; we don't want to address the hardest part of a problem at the point with the smallest range of design options. Of course, one could take the fashionable approach and ***refactor*** the design; however, designing, building, and incrementally validating a system only to change it because risky areas were not proactively considered is hardly efficient and is not CbyC.

Managing risk must be done on a case-by-case basis. Common examples are prototyping of a user interface to ensure its acceptability, performance modeling to ensure the selected high-level design can provide adequate throughput, and early integration of complex external hardware devices.

We can capture this idea with the sound bite ***do the hard things first***.

Thinking Hard!

All of the ideas above are supplemented by a culture of “thinking hard” about the real objectives of the system being developed and using the right tools for each job. CbyC stresses that *logical reasoning* should demonstrate the fitness for purpose of a system . A system developed in this way should be certifiable to any applicable safety or security standard. A “box ticking” mentality focused on doing specific things called for in a standard or prescriptive development process document will not achieve the desired result. Similarly, slavish devotion to large, integrated tool sets and their accompanying implicitly defined development methods is also unproductive. Sometimes, the right tool for the job is *grep* and sometimes it is a *theorem prover*; one size does not fit all.

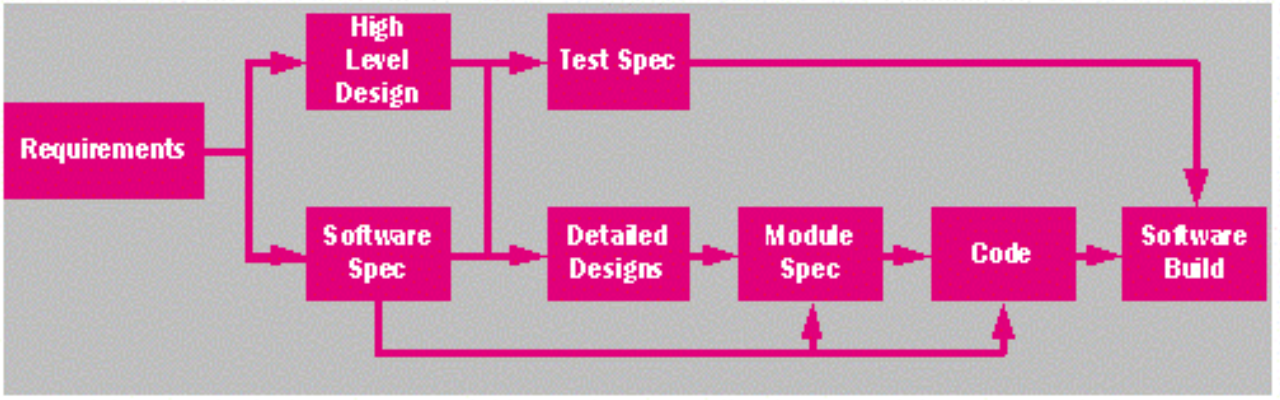
We say ***argue your corner*** and ***screws? use a screwdriver, not a hammer***.

Process Steps

So, how do we marshal the above building blocks into a coherent development process? Figure 1 is a simplified diagram of the core CbyC process. The simplifications are

- removal of much of the parallelism and overlap between process stages
- omission of feedback from the various validation activities that can clearly affect any earlier deliverable and cause re-entry to any previous activity

Figure 1. Core correctness by construction process



Key Elements

The ***requirements*** describe the purpose of the software, the functions it must provide, and the non-functional requirements such as security, safety, and performance. Relevant facts about the application ***domain*** also must be captured (for reasons described under specification below).

Requirements for secure systems include the security target, which will be stated in English language. For high assurance levels, a ***formal security policy model*** also is written. This formalizes the technical aspects of the security target and has two benefits:

1. It makes the security target absolutely precise.
2. It allows more rigorous validation of subsequent deliverables.

The ***software specification*** is a complete (including error conditions) and precise description of the behavior of the software viewed as a black box. It contains no information about the software’s internal structure.

CbyC makes a clear distinction between requirements (i.e., the change we are trying to effect) and specification (the behavior of the machine that will create that change). A key element connecting the two is knowledge about the domain in which the system will operate. More formally we say

$$D, S \# R$$

or, *the machine specified by S, in the domain D, satisfies the requirement R.*

The ***high-level design*** describes the architecture of the software, where key non-functional properties such as safety and security are addressed. It is also the point at which we provide for irresolvable requirements uncertainties by selecting a design that is flexible in areas of probably change. Rather unintuitively, CbyC's response to requirements uncertainty is *more* design not *less*.

A number of ***detailed designs*** describe, where the semantic gap between the software specification and code would otherwise be too large, the operation of different aspects of the software (e.g., process structure, database schema).

Module specifications define the state and behavior encapsulated by each software module. Module design is primarily driven by considerations of ***information flow.*** Considering each design decision in terms of what information flows it will generate and striving to minimize such flows will result in a design that exhibits the desirable properties of low coupling and high cohesion.

Code refers to the executable code of each module. Where possible, SPARK will be used (although pragmatic use of other languages will be made where appropriate), and the code will have been carefully, statically analyzed to eliminate certain classes of errors. Where appropriate, code review is done *after* static analysis. Compilation is a very small part of the development process that also comes after static analysis. Programmers *do not* perform informal testing; the combination of strong specifications, unambiguous languages, and strong static analysis makes it feasible to delay testing until each build is integrated.

The ***test specification*** is obtained primarily from the software specification, together with the requirements and the high-level design. Boundary value analysis is used to generate tests that cover the specification, which are supplemented with tests for behavior introduced by the design but not visible in the specification. In addition, tests for non-functional requirements are generated directly from the requirements document. Because of the strong validation and small semantic gaps inherent in the CbyC approach, it is not useful to perform code-based, white-box, or unit testing; all testing in CbyC is system level and specification driven.

Each ***build*** is a version of the software that offers a subset of its behavior. Typically, early builds include only infrastructure software and have little application functionality. Each build acts as a test harness for subsequent code. Although CbyC omits unit testing, it does measure code coverage as system tests are executed. When gaps are found in coverage, one of three things can be done:

1. Usually the gap is caused by code, which implements some aspect of the design not visible at the specification level. In that case, suitable tests are added.
2. Sometimes the gap reflects unnecessary code, and the code is removed.
3. Sometimes the code cannot be reached by normal operation of the system but is still necessary (e.g., defensive code). In that case, and only then, unit tests are written at the module level.

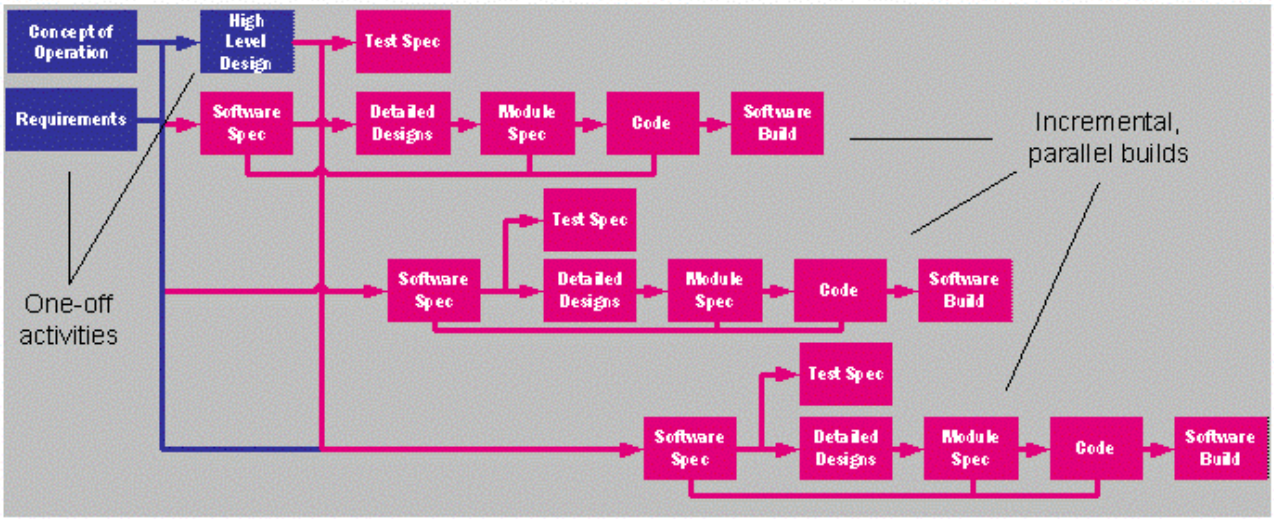
The ***installed software*** is the final build, configured and installed in its operational environment. The final build is no different from any of the intermediate builds that precede it, except that it includes all of the planned functionality. This form of continuous integration and continuous validation is a key way in which CbyC avoids late-breaking problems (i.e., ***integration bottleneck***).

Parallelism

Although Figure 1 appears to describe a largely sequential process, a significant degree of parallel activity is possible in practice. Parallelism is possible in three areas.

1. Two different kinds of independent activity can be done in parallel. For example, the high-level design is based largely on non-functional requirements and does not depend on details of the functional specification, so it can be done in parallel with the software specification.
2. Areas in which the system can be partitioned can be developed in parallel. They may simultaneously progress at different or the same stages of development.
3. Incremental builds allow for testing of one build to be done in parallel with coding of the subsequent build. Figure 2 shows this form of parallelism.

Figure 2. Parallelism introduced by incremental build development



Feedback

Although CbyC aims to prevent the introduction of as many errors as possible, some errors are inevitably made and are revealed by the various forms of strong validation used after each activity. Where errors are discovered, simply fixing them at the point of discovery is not the appropriate response. Point fixes may compromise specification and design decisions on which other parts of the system may depend and will, as a result, introduce more problems later on. Instead, CbyC handles each identified defect as follows.

Root cause analysis

First, we iterate back to the point where the defect was introduced. Because CbyC aims to discover each defect close to the point of its introduction, this iteration should not go back too far. In a less satisfactory case, however, we might have found a requirements error during coding. We correct the error *at the point where it was introduced* and then *rework all subsequent deliverables*. Although this sounds like an alarming amount of rework, it is feasible because the CbyC process minimizes the number of errors requiring this treatment and because the avoidance of repetition minimizes the number of documents requiring change. By fixing the root cause and reworking dependent items, we ensure that all system documentation remains consistent (which is essential to simplifying certification) and identify parts of the system affected by the error (rather than rediscovering it in several places).

In practice, the rework from error correction is batched and accommodated in the iterative build process.

Process improvement

As well as correcting the root cause of the error, we also seek to understand *how* the error was made. We then improve the development process to prevent errors of that kind from reoccurring (remembering that error *prevention* is the primary goal of CbyC). We might, for example, introduce a new review checklist item or use an additional tool.

Flexibility

CbyC is not a single, rigid process; it is a framework and set of principles. We tailor every project based on its nature and criticality. Projects may differ in many aspects:

1. **Level of rigor**. Some projects require fully formal proofs of correspondence between formal specifications; others may not justify any formality at all. Note, however, that a minimum level of rigor exists, which is worthwhile regardless of the criticality of the system being produced. We routinely use SPARK on non-critical systems and almost invariably carry out a proof of absence of run-time errors even if we do no other strong validation or proof. Our experience is that doing so reduces cost overall.

2. **Techniques/notations** at each stage. Different kinds of software require different notations (e.g., embedded systems need a very different style of specification from database applications).
3. **Subsets of activities**. Some projects may omit some of the activities or add extra activities.
4. **Content of design**. The amount of detail in the design depends on the system's size and complexity.
5. **Formality of evaluation**. The amount of evidence collected and the rigor with which it is controlled can be adapted according to how rigorously the software will be evaluated. The process can develop software to the highest levels of safety (e.g., safety integrity level 4 as defined by UK MoD DEF STAN 00-56) and security (e.g., Common Criteria assurance level EAL 7).

Generic Activities

Naturally, the whole CbyC process is supported by generic activities that are not project specific. The following must be addressed.

1. Process planning
2. Staff training and competency
3. Traceability from requirements through specification to code and test cases
4. Fault management
5. Change management
6. Configuration management
7. Metrics collection

Conclusions

There are no silver bullets, and CbyC makes no claims to change that stark fact. It is, however, clear that careful use of the best currently available tools and techniques can radically improve software development performance.

CbyC combines rigorous, mathematically based notations with agile approaches to incremental development; the result is industry beating low defect rates combined with high productivity.

It isn't magic, just good engineering.

References

[Hall 02]	Hall, A. & Chapman, R. "Correctness By Construction: Developing a Commercial Secure System[9]." *IEEE Software 19*, 1 (Jan-Feb 2002): 18-25.
[Barnes 06]	Barnes, J.; Chapman, R.; Johnson, R.; Widmaier, J.; Cooper, D.; & Everett, W. "Engineering the Tokeneer Enclave Protection Software[10]." *Proceedings of the IEEE International Symposium on Secure Software Engineering (ISSSE)*, 2006.

Praxis High Integrity Systems Ltd. Copyright